

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

**Code Integrity Verification That Includes One Or More
Cycles**

Inventor(s):

**Mariusz H. Jakubowski
Ramarathnam Venkatesan
Yacov Yacobi**

ATTORNEY'S DOCKET NO. MS1-527US

1
2 **RELATED APPLICATIONS**

3 This application claims the benefit of U.S. Provisional Application No.
4 60/199,622, filed April 25, 2000, entitled "Cyclic Verification of Code Integrity"
5 to Mariusz H. Jakubowski, Ramarathnam Venkatesan, and Yacov Yacobi.
6

7 **TECHNICAL FIELD**

8 This invention relates to protecting digital objects, such as software.
9

10 **BACKGROUND**

11 Digital content (e.g., software, firmware, etc.) are often distributed to
12 consumers via fixed computer readable media, such as a compact disc (CD-ROM),
13 digital versatile disc (DVD), soft magnetic diskette, or hard magnetic disk (e.g., a
14 preloaded hard drive). More recently, more and more content is being delivered in
15 digital form online over private and public networks, such as Intranets and the
16 Internet. Online delivery improves timeliness and convenience for the user, as
17 well as reduces delivery costs for a publisher or developers. Unfortunately, these
18 worthwhile attributes are often outweighed in the minds of the
19 publishers/developers by a corresponding disadvantage that online information
20 delivery makes it relatively easy to obtain pristine digital content and to pirate the
21 content at the expense and harm of the publisher/developer.

22 One concern of the publisher/developer is the ability to check digital
23 content, after distribution, for alteration. Such checking, is often referred to as
24 SRI (Software Resistance to Interference). The desire to check for such alterations
25

1 can vary (e.g., to ensure that the content continues to operate as intended by the
2 publisher/developer, to protect against improper copying, etc.).

3 The unusual property of content is that the publisher/developer (or reseller)
4 gives or sells the *content* to a client, but continues to restrict *rights* to use the
5 content even after the content is under the sole physical control of the client. For
6 instance, a software developer typically sells a limited license in a software
7 product that permits a user to load and run the software product on one or more
8 machines (depending upon the license terms), as well as make a back up copy.
9 The user is typically not permitted to make unlimited copies or redistribute the
10 software to others.

11 Such scenarios reveal a peculiar arrangement. The user that possesses the
12 digital bits often does not have full rights to their use; instead, the provider retains
13 at least some of the rights. In a very real sense, the legitimate user of a computer
14 can be an adversary of the content provider.

15 One of the uses for SRI is to provide "digital rights management" (or
16 "DRM") protection to prevent unauthorized distribution of, copying and/or illegal
17 operation of, or access to the digital content. An ideal digital content distribution
18 system would substantially prevent unauthorized distribution/use of the digital
19 content. Digital rights management is fast becoming a central requirement if
20 online commerce is to continue its rapid growth. Content providers and the
21 computer industry must quickly address technologies and protocols for ensuring
22 that digital content is properly handled in accordance with the rights granted by
23 the developer/publisher. If measures are not taken, traditional content providers
24 may be put out of business by widespread theft or, more likely, will refuse
25 altogether to deliver content online.

Various DRM techniques have been developed and employed in an attempt to thwart potential pirates from illegally copying or otherwise distributing the digital goods to others. For example, one technique includes requiring or otherwise encouraging the consumer to register the digital content with the provider, for example, either through the mail or online via the Internet or a direct connection. Thus, the digital content may require the consumer to enter a registration code before allowing the digital content to be fully operational or fully accessed. Unfortunately, such techniques are not always effective since unscrupulous individuals/organizations need only break through or otherwise undermine the protections in a single copy of the digital content. Once broken, copies of the digital good can be illegally distributed, hence such techniques are considered to be Break-Once, Run-Everywhere (BORE) susceptible.

Accordingly, there remains a need for a technique that addresses the concerns of the publisher/developer, allowing alteration of the digital content to be identified to assist in protecting the content from many of the known and common attacks, but does not impose unnecessary and burdensome requirements on legitimate users.

SUMMARY

Code integrity verification that includes one or more cycles is described herein.

According to one aspect, an object to be protected is separated into multiple modules, any one or more of which can include a checkpoint and corresponding checkpoint value that can be used to verify the integrity of any one or more other modules. Each module is then separated into multiple blocks, and a message

1 authentication code (MAC) value of each of these modules is computed (based on
2 the blocks within the module). The computed module MAC values are then
3 incorporated into selected ones of the multiple modules (referred to here as
4 "checker modules"). Incorporating the calculated MAC values into the checker
5 modules will alter the content of the checker modules, and thus alter the
6 previously calculated MAC values for those checker modules. Thus, a new MAC
7 value for each of the checker modules is calculated. For each of the checker
8 modules, a new block is also added having content that results in restoring the
9 MAC value of the checker module back to its original value (the content for the
10 new block is determined based on the MAC function and the new MAC values).
11 Thus, the checker modules can be subsequently verified based on the MAC values
12 stored in other modules.

13 14 **BRIEF DESCRIPTION OF THE DRAWINGS**

15 The same numbers are used throughout the drawings to reference like
16 elements and features.

17 Fig. 1 illustrates an exemplary distribution architecture in which objects
18 (e.g., software or firmware code) are transformed into protected digital objects and
19 distributed in their protected form.

20 Fig. 2 shows a general example of a computer that can be used in
21 accordance with certain embodiments of the invention.

22 Fig. 3 shows an exemplary integrity verification tool implemented by a
23 production server in more detail.
24
25

Fig. 4 shows an exemplary protected object as shipped to a client, and illustrates control flow through the object as a client-side evaluator evaluates the object for any sign of tampering.

Fig. 5 shows another exemplary protected object partitioned into multiple modules.

Fig. 6 illustrates an exemplary data flow for an integrity verification tool in generating checkpoint values.

Fig. 7 is a flowchart illustrating an exemplary process for generating cyclic checkpoint values in accordance with certain embodiments of the invention.

DETAILED DESCRIPTION

A digital content distribution architecture produces and distributes content (also referred to as objects) in a fashion that provides resistance to interference with the content. The content can be made resistant to interference in any of a wide variety of situations, such as any time the publisher/developer of the content desires to make it difficult to alter the content. The distribution architecture provides such resistance to the content by manipulating the content to allow portions to include information that can be used to verify the integrity of other portions (e.g., to verify that they have not been altered). This architecture is particularly useful for verifying the integrity of code, such as software code or firmware code. For discussion purposes, many of the examples are described in the context of software code, although the techniques described herein are effective for non-software code as well.

Distribution Architecture

Fig. 1 illustrates an exemplary distribution architecture 100 in which objects (e.g., software or firmware code) are transformed into protected digital objects and distributed in their protected form. One specific example of architecture 100 is a digital rights management (DRM) distribution architecture that renders the content resistant to many known forms of attack. Architecture 100 has a system 102 that develops or otherwise produces the protected object and distributes the protected object to a client 104 via some form of distribution channel 106. The protected digital objects may be distributed in many different ways. For instance, the protected digital objects may be stored on a computer-readable medium 108 (e.g., CD-ROM, DVD, floppy disk, etc.) and physically distributed in some manner, such as conventional vendor channels or mail. The protected objects may alternatively be downloaded over a network (e.g., the Internet) as content or files 110.

Developer/producer system 102 has a memory 112 to store an original object 114, as well as protected object 116 created from the original object. System 102 also has a production server 118 that transforms original object 114 into protected object 116 that is suitable for distribution. Production server 118 has a processing system 120 and implements an integrity verification tool 122. Generally speaking, integrity verification tool 122 automatically parses original object 114 and modifies object 114 for cyclic integrity verification into the code to produce protected object 116.

Original object 114 represents the code as originally produced, without any protection or code modifications. Protected object 116 is a unique version of the software product or data after the cyclic integrity verification has been inserted

1 into the code. Protected object 116 is functionally equivalent to and derived from
2 original object 114, but is modified to enable the client to determine whether the
3 product has been tampered with.

4 Developer/producer system 102 is illustrated as a single entity, with
5 memory and processing capabilities, for ease of discussion. In practice, however,
6 system 102 may be configured as one or more computers that jointly or
7 independently perform the tasks of transforming the original object into the
8 protected object.

9 Client 104 has a processor 124, memory 126 (e.g., RAM, ROM, Flash, hard
10 disk, CD-ROM, etc.), one or more input devices 128 (e.g., keyboard, joystick,
11 voice recognition, etc.), and one or more output devices 130 (e.g., monitor,
12 speakers, etc.). The client may be implemented as a general purpose computing
13 unit (e.g., desktop PC, laptop, etc.) or as other devices, such as set-top boxes,
14 audio/video appliances, game consoles, and the like. Processor 124 can optionally
15 be a "secure" processor that supports various security features, such as a certificate
16 for use in cryptography, a unique processor id or serial number, etc.

17 Client 104 runs an operating system 132, which is stored in memory 126
18 and executed on processor 124. Operating system 132 represents any of a wide
19 variety of operating systems, such as a multi-tasking, open platform system (e.g., a
20 "Windows"-brand operating system from Microsoft Corporation). The operating
21 system 132 includes an evaluator 134 that evaluates the protected objects to
22 determine whether the protected objects have been tampered with or modified in
23 any manner. This evaluation can be performed at different times, such as prior to
24 loading and execution, during execution, etc., and can be based on the shape of the
25 code and/or the behavior of the code. Evaluators 134 on different clients 104 can

1 be configured to perform this evaluation at different times, or alternatively an
2 evaluator 134 on a single client 104 can be configured to perform this evaluation
3 at different times (e.g., prior to loading and execution as well as during execution).

4 The protected object may have multiple security checks that operate to
5 ensure that client 104 is authorized to execute the protected object. By way of
6 example, a check for the existence of a secret key(s) within the object, for the
7 existence of a CD-ROM, for the existence of a watermark, etc. If a pirate or other
8 malicious user were to attempt to modify the code so that such a check were not
9 performed, the code integrity verification process would detect that the code had
10 been modified and take appropriate action in response.

11 Some protection schemes involve executing instructions, analyzing data,
12 and performing other tasks in the most secure areas of the operating system 132
13 and processor 124. Accordingly, the evaluator 134 includes code portions that
14 may be executed in these most secure areas of the operating system and secure
15 processor. Although the evaluator 134 is illustrated as being integrated into the
16 operating system 132, it may be implemented separately from the operating
17 system.

18 In the event that the client detects some tamper activity, processor 124
19 acting alone, or together with operating system 132, may decline to execute the
20 suspect digital object. For instance, the client may determine that the software
21 product is an illicit copy because the evaluations performed by evaluator 134 are
22 not successful. In this case, the evaluator 134 informs processor 124 and/or
23 operating system 132 of the suspect code and processor 124 may decline to run
24 that software product, if already running then operating system 132 or processor
25

1 124 may terminate execution of the software product, operating system 132 may
2 notify an administrator, etc.

3 It is further noted that operating system 132 may itself be the protected
4 object. That is, operating system 132 may be modified to implement a code
5 integrity verification process to produce a product that makes it easy to detect such
6 copying. In this case, processor 124 may be configured to detect an improper
7 version of the operating system during the boot process (or at other times) and
8 prevent the operating system from fully or partially executing and obtaining
9 control of system resources.

10 Alternatively, a protected object itself may perform the evaluation rather
11 than evaluator 134. In this situation, the protected object includes one or more sets
12 of instructions that can be executed to verify its own integrity, and then inform
13 processor 124 and/or operating system 132 in the event of an integrity verification
14 failure.

15 For protected objects delivered over a network, the client 104 implements
16 tamper-resistant software (not shown) to connect to server 102 using an SSL
17 (secure sockets layer) or other secure and authenticated connection to purchase,
18 store, and utilize the object. The objected may be encrypted using well-known
19 algorithms (e.g., RSA) and compressed using well-known compression techniques
20 (e.g., ZIP, RLE, etc.).

21 Fig. 2 shows a general example of a computer 142 that can be used in
22 accordance with certain embodiments of the invention. Computer 142 is shown as
23 an example of a computer that can perform the functions of developer/producer
24 system 102 or client 104 of Fig. 1.

Computer 142 includes one or more processors or processing units 144, a system memory 146, and a bus 148 that couples various system components including the system memory 146 to processors 144. The bus 148 represents one or more of any of several types of bus structures, including a memory bus or memory controller, a peripheral bus, an accelerated graphics port, and a processor or local bus using any of a variety of bus architectures. The system memory includes read only memory (ROM) 150 and random access memory (RAM) 152. A basic input/output system (BIOS) 154, containing the basic routines that help to transfer information between elements within computer 142, such as during start-up, is stored in ROM 150.

Computer 142 further includes a hard disk drive 156 for reading from and writing to a hard disk, not shown, connected to bus 148 via a hard disk driver interface 157 (e.g., a SCSI, ATA, or other type of interface); a magnetic disk drive 158 for reading from and writing to a removable magnetic disk 160, connected to bus 148 via a magnetic disk drive interface 161; and an optical disk drive 162 for reading from or writing to a removable optical disk 164 such as a CD ROM, DVD, or other optical media, connected to bus 148 via an optical drive interface 165. The drives and their associated computer-readable media provide nonvolatile storage of computer readable instructions, data structures, program modules and other data for computer 142. Although the exemplary environment described herein employs a hard disk, a removable magnetic disk 160 and a removable optical disk 164, it should be appreciated by those skilled in the art that other types of computer readable media which can store data that is accessible by a computer, such as magnetic cassettes, flash memory cards, digital video disks, random access

1 memories (RAMs) read only memories (ROM), and the like, may also be used in
2 the exemplary operating environment.

3 A number of program modules may be stored on the hard disk, magnetic
4 disk 160, optical disk 164, ROM 150, or RAM 152, including an operating system
5 170, one or more application programs 172, other program modules 174, and
6 program data 176. A user may enter commands and information into computer
7 142 through input devices such as keyboard 178 and pointing device 180. Other
8 input devices (not shown) may include a microphone, joystick, game pad, satellite
9 dish, scanner, or the like. These and other input devices are connected to the
10 processing unit 144 through an interface 168 (e.g., a serial port interface) that is
11 coupled to the system bus. A monitor 184 or other type of display device is also
12 connected to the system bus 148 via an interface, such as a video adapter 186. In
13 addition to the monitor, personal computers typically include other peripheral
14 output devices (not shown) such as speakers and printers.

15 Computer 142 optionally operates in a networked environment using
16 logical connections to one or more remote computers, such as a remote computer
17 188. The remote computer 188 may be another personal computer, a server, a
18 router, a network PC, a peer device or other common network node, and typically
19 includes many or all of the elements described above relative to computer 142,
20 although only a memory storage device 190 has been illustrated in Fig. 2. The
21 logical connections depicted in Fig. 2 include a local area network (LAN) 192 and
22 a wide area network (WAN) 194. Such networking environments are
23 commonplace in offices, enterprise-wide computer networks, intranets, and the
24 Internet. In the described embodiment of the invention, remote computer 188
25 executes an Internet Web browser program (which may optionally be integrated

1 into the operating system 170) such as the "Internet Explorer" Web browser
2 manufactured and distributed by Microsoft Corporation of Redmond, Washington.

3 When used in a LAN networking environment, computer 142 is connected
4 to the local network 192 through a network interface or adapter 196. When used
5 in a WAN networking environment, computer 142 typically includes a modem 198
6 or other component for establishing communications over the wide area network
7 194, such as the Internet. The modem 198, which may be internal or external, is
8 connected to the system bus 148 via an interface (e.g., a serial port interface 168).
9 In a networked environment, program modules depicted relative to the personal
10 computer 142, or portions thereof, may be stored in the remote memory storage
11 device. It is to be appreciated that the network connections shown are exemplary
12 and other means of establishing a communications link between the computers
13 may be used.

14 Computer 142 also optionally includes one or more broadcast tuners 200.
15 Broadcast tuner 200 receives broadcast signals either directly (e.g., analog or
16 digital cable transmissions fed directly into tuner 200) or via a reception device
17 (e.g., via antenna 110 or satellite dish 114 of Fig. 1).

18 Generally, the data processors of computer 142 are programmed by means
19 of instructions stored at different times in the various computer-readable storage
20 media of the computer. Programs and operating systems are typically distributed,
21 for example, on floppy disks or CD-ROMs. From there, they are installed or
22 loaded into the secondary memory of a computer. At execution, they are loaded at
23 least partially into the computer's primary electronic memory. The invention
24 described herein includes these and other various types of computer-readable
25 storage media when such media contain instructions or programs for implementing

the steps described below in conjunction with a microprocessor or other data processor. The invention also includes the computer itself when programmed according to the methods and techniques described below. Furthermore, certain sub-components of the computer may be programmed to perform the functions and steps described below. The invention includes such sub-components when they are programmed as described. In addition, the invention described herein includes data structures, described below, as embodied on various types of memory media.

For purposes of illustration, programs and other executable program components such as the operating system are illustrated herein as discrete blocks, although it is recognized that such programs and components reside at various times in different storage components of the computer, and are executed by the data processor(s) of the computer.

Fig. 3 shows an exemplary integrity verification tool 122 implemented by production server 118 of Fig. 1 in more detail. Integrity verification tool 122 is configured to transform an original object 114 into a protected object 116. The transformation process is usually applied just before the object is released to manufacture or prior to being downloaded over a network. The process is intended to produce an object that is protected from various forms of attack and illicit copying activities. Integrity verification tool 122 may be implemented in software, firmware, hardware, or a combination thereof.

Integrity verification tool 122 includes an analyzer 210 that analyzes the original object 114 and parses it into multiple segments (also referred to herein as modules). Analyzer 210 may attempt to intelligently segment the object along natural boundaries inherent in the object, such as by parsing the code according to

1 logical groupings of instructions, such as routines, or sub-routines, or instruction
2 sets. Alternatively, analyzer 210 may segment the object without regard for the
3 natural boundaries, such as randomly or by using one or more programmed (or
4 pre-determined) segment sizes.

5 In one specific implementation for analyzing software code, analyzer 210 is
6 configured as a software flow analysis tool that converts the software program into
7 a corresponding flow graph. The flow graph is partitioned into many clusters of
8 nodes. The segments may then take the form of sets of one or more nodes in the
9 flow graph. For more information on this technique, the reader is directed to co-
10 pending U.S. Patent Application Serial Number 09/525,694, entitled "A
11 Technique for Producing, Through Watermarking, Highly Tamper-Resistant
12 Executable Code and Resulting "Watermarked" Code So Formed", which was
13 filed March 14, 2000, in the names of Ramarathnam Venkatesan and Vijay
14 Vazirani. This Application is assigned to Microsoft Corporation and is hereby
15 incorporated by reference.

16 Integrity verification tool 122 also includes a checkpoint calculator/insertor
17 212 that generates and inserts checkpoint values into the segments or modules of
18 the object. As discussed in more detail below, the checkpoint value for a
19 particular module is a verification value (also referred to as a checksum value)
20 based on the content of that module and may be modified due to checkpoint values
21 for other modules. Checkpoint calculator/insertor 212 outputs the protected object
22 116 that is ready for mass production and/or distribution.

23 Checkpoints and corresponding checkpoint values can be inserted into
24 segments of the code to verify the integrity of any one or more segments of the
25 code. This ability to verify the integrity of any segment can result in cycles of

1 integrity verification being created. For example, segment A may verify the
2 integrity of segment B, which in turn verifies the integrity of segment A. One or
3 more such cycles can be established within the code, and segments can belong to
4 multiple cycles. Continuing with the previous example, segment B may also
5 verify the integrity of segment C, which in turn verifies the integrity of segment B
6 (and may or may not also verify the integrity of segment A).

7 Integrity verification tool 122 may further be configured with a quantitative
8 unit 214 that enables a producer/developer to define how much protection should
9 be applied to the original object. For instance, the producer/developer might elect
10 to set the number of checkpoints (e.g., 500 or 1000) added to the object as a result
11 of the protection, or define a maximum number of lines/bytes of code that are
12 added for protection purposes. Quantitative unit 214 may include a user interface
13 (not shown) that allows the user to enter parameters defining a quantitative amount
14 of protection.

15 Quantitative unit 214 provides control information to analyzer 210 and
16 checkpoint calculator/insertter 212 to ensure that these components satisfy the
17 specified quantitative requirements. Suppose, for example, the
18 producer/developer enters a predefined number of checkpoints (e.g., 500). With
19 this parameter, analyzer 210 ensures that there are a sufficient number of segments
20 (e.g., at least 500), and checkpoint calculator/insertter 212 ensures that the resulting
21 number of checkpoints approximates 500.

22 In some situations, care should be taken in selecting the number of
23 checkpoints that are included in an object. If the number of checkpoints becomes
24 too large, then various statistical indicators of the object's operation (e.g., payload
25 control graphs, data graphs, etc.) can become abnormal. Such abnormalities can

1 provide indicators to malicious users as to where certain checkpoints are located
2 within an object and assist them in removing such checkpoints.

3 Fig. 4 shows an exemplary protected object 116 as shipped to the client,
4 and illustrates control flow through the object as client-side evaluator 134
5 evaluates the object 116 for any sign of tampering. The protected object 116 has
6 multiple checkpoints 230(1), 230(2),..., 230(N) spread throughout the object.
7 When executing the object 116, evaluator 134 passes through the various
8 checkpoints 230(1)-230(N) to determine whether the checks are valid, thereby
9 verifying the authenticity of the protected object.

10 If any checkpoint fails, the client is alerted that the object may not be
11 authentic. In this case, the client may refuse to execute the object or disable
12 portions of the object in such a manner that renders it relatively useless to the user.

13 Fig. 5 shows another exemplary protected object 116 partitioned into
14 multiple segments or modules. Although object 116 may have a large number of
15 modules, for ease of explanation only three modules 250, 252, and 254 are shown.
16 A verification cycle is created in object 116, with module 250 verifying module
17 254, module 254 verifying module 252, and module 252 verifying module 250.
18 One module verifying another module refers to the one module (also referred to
19 herein as a "checker" module) including both a checkpoint that identifies which
20 module is to have its integrity verified and a checkpoint value corresponding to
21 that module. The checkpoint value is a value that is inserted into the checker
22 module by integrity verification tool 122 of Fig. 1 and that can be re-calculated by
23 evaluator 134, as discussed in more detail below.

24 Additional verification is also performed, with module 254 also verifying
25 module 250 and module 252 also verifying module 254. Any module in object

1 116 can verify the integrity of (be a checker module for) any other module
2 (including itself) in the object. As the number of checks (verifications) increases,
3 so to does the difficulty faced by a malicious user in removing all of the checks
4 increase. In order to verify the integrity of protected object 116, each of these
5 integrity verifications should be satisfied.

6 Each module of a protected object can check any one or more other
7 modules in the object. This allows for a very large number of checks within the
8 object. However, if each module checks at most only one other module, then the
9 number of checks would not exceed k checks, where k represents the number of
10 modules in the object. In one implementation, the number of checks should not
11 exceed the exponential value (exp) of k (e^k), where k represents the number
12 modules in the object and e is the constant e (the base of the natural logarithm).

13 Verification of the integrity of a particular module occurs at a checkpoint.
14 In the illustrated example, each module 250 – 254 that is verifying the integrity of
15 another module includes a checkpoint (CP) for each module it is verifying. Each
16 checkpoint also has a corresponding checkpoint value (V), which is the value that
17 is obtained by applying a cryptographic message authentication code (MAC) or
18 hash function to the module being verified. This value is included in the module
19 performing the verification along with the checkpoint. For example, module 252
20 includes a checkpoint 256 and checkpoint value 258 for module 250, as well as a
21 checkpoint 260 and checkpoint value 262 for module 254.

22 The checkpoints and checkpoint values can be located anywhere within the
23 module. They may be stored entirely within one block (discussed in more detail
24 below) or alternatively separated over multiple blocks. Additionally, the
25

1 checkpoint and checkpoint value may be stored together (e.g., as illustrated in
2 module 254) or alternatively separate (e.g., as illustrated in module 252).

3 Using a MAC or hash function to generate a checkpoint value for a module
4 being verified allows the integrity of the module to be verified because any change
5 (e.g., even of one byte) by a malicious user will result in a different checkpoint
6 value (and thus any changes by the malicious user detected). However, this is
7 troublesome for cyclic verification as illustrated in Fig. 5, because the verification
8 of one module relies on the addition of a checkpoint value in another module. For
9 example, assume that the checkpoint value for module 254 is generated and stored
10 in module 252, then the checkpoint value for module 252 is generated and stored
11 in module 250, and then the checkpoint value for module 250 is generated and
12 stored in module 254. This would change the checkpoint value generated for
13 module 254 as stored in module 252, which would change the checkpoint value
14 generated for module 252 and stored in module 250, etc.

15 As discussed in more detail below, such cyclic verification can be
16 performed by setting aside and using some "free" space inside modules. This
17 space is part of the code bytes verified by the checkpoint value computation. If a
18 particular checkpoint value is incorrect, the code bytes stored in the extra space
19 can be changed until the checkpoint value becomes proper.

20 Fig. 6 illustrates an exemplary data flow for integrity verification tool 122
21 of Fig. 1 in generating checkpoint values. The checkpoint value generation
22 process of Fig. 6 is described with reference to Fig. 5 and specifically with
23 reference to modules 250 and 254. It is to be appreciated, however, that the
24 process is repeated for each checkpoint value being generated.
25

Multiple blocks (e.g., 64-bit blocks) are identified in each module 250 and 254. Every bit in the module may be included in one or more blocks, or alternatively some portions of the module may not be included in any block. These blocks can be overlapping or non-overlapping (or a combination thereof). The number of blocks in a module can vary between different objects and between different modules within the same object, and the size of blocks can vary between different objects, between different modules, and between blocks within the same module. One or more of these blocks in module 250 includes the checkpoint 264 indicating to verify module 254 and the checkpoint value 266 indicating the value that should be generated based on module 254. Similarly, one or more blocks in module 254 includes the checkpoint 268 indicating to verify module 250 and the checkpoint value 270 indicating the value that should be generated based on module 250.

Generation of the checkpoint value 270 for module 250 is performed based on the individual blocks of module 250. The blocks of module 250 are accessed individually by integrity verification tool 122 and an exclusive-or (xor) operator 300 performs a bit-by-bit xor operation based on the bits in the module and a set of bits received from an encryption process 302 (discussed in more detail below). Integrity verification tool 122 can access the blocks in any order (e.g., in the order they appear in module 250, a pseudo-random order, etc.), so long as that order is known by (or made known to) evaluator 134.

The output 304 of xor operator 300 is input to a delay buffer 306, which in turn outputs the delayed value 308 to encryption process 302. Encryption process 302 is any of a variety of well-known symmetric block ciphers (such as DES, LC4, LC5, etc.) that operates based on a secret key 310. The output 312 of

1 encryption process 302 is also input to xor operator 300. Exclusive-or operator
2 300 thus generates an exclusive-or output based on a block of message 250 and the
3 previous output of operator 300 (after being encrypted). For the first block of
4 message 250, process 302 outputs a value known to both tool 122 and evaluator
5 134 (e.g., all 0's, all 1's, alternating 0 and 1, etc.). The output 304 of operator 300
6 when the final block of module 250 is input to operator 300 is used as the original
7 MAC value (also referred to as the original hash value or original checkpoint
8 value) for module 250.

9 This process is then repeated for module 254, generating an original MAC
10 value for module 254. After the original MAC values for each of the modules 250
11 and 254 is generated, the original MAC value for module 250 is stored in module
12 254 as checkpoint value 270 and the original MAC value for module 254 is stored
13 in module 250 as checkpoint value 264. This storage of the original MAC values
14 occurs after the original MAC values for each module are generated.

15 In the illustrated example of Fig. 5, the checkpoint values 266 and 270 are
16 stored within blocks of modules 250 and 254, respectively, that are used in
17 generating the original MAC values. For generation of the original MAC values,
18 default values (e.g., all 0's or all 1's) can be stored in these areas and then
19 overwritten when the original MAC values are stored. Alternatively, an additional
20 block(s) could be added to each of the modules and the original MAC values
21 stored in these added blocks.

22 Integrity verification tool 122 then repeats the checkpoint value generation
23 process with the newly modified (to include the original MAC values) modules
24 250 and 254, resulting in new MAC values for both modules 250 and 254. As the
25 content of module 250 has changed, the new MAC values will differ from the old

An additional block 272 is added to module 250 and an additional block 274 is added to module 254. Although blocks 272 and 274 are illustrated as being added to the end of the module, alternatively they may be located in any of a variety of locations. For example, an additional block may be added to the beginning of the module (e.g., block 276 of module 252) or elsewhere within module 252.

The content for the additional offset block can be generated in a variety of different manners. In one implementation, the value is generated by inputting the new MAC value into encryption process 302 (e.g., via delay element 306), where the new MAC value is encrypted based on secret key 310. The encrypted value is then input to xor operator 312 along with the original MAC value. Operator 312 performs a bit-by-bit exclusive-or operation of these two inputs, which results in an output block 314. Output block 314 identifies the content of the additional

1 block that is to be added to the module. Alternatively, other processes could be
2 used to identify the content of the additional block, such as trial and error, etc.

3 Care should be taken in the selection of content for the additional block so
4 that the added block does not alter the functionality of the module (and thus of the
5 object). By way of example, the additional block could be a data block, a No
6 Operation instruction, a Jump instruction to the next instruction, or some other
7 instruction(s) that does not alter the functionality of the module. In some
8 situations (e.g., use of a Jump instruction to jump over the added content), the
9 additional instruction(s) (e.g., Jump) are added prior to determining the content for
10 the additional block (and prior to generating the new checkpoint value), so that the
11 additional instruction(s) are part of the change to the module that is compensated
12 for by the additional block. These additional instructions can be added, for
13 example, as more block(s) in the module (in addition to the additional block added
14 to compensate for changes to the module).

15 Now also referring to Fig. 1, when the protected content is subsequently
16 transferred to client 104 via distribution channel 106, the verification process
17 performed by evaluator 134 is similar to that performed by integrity verification
18 tool 122. To verify module 250, evaluator 134 starts with the first block of module
19 250, inputting the block to an operator 300. The resultant value is then delayed
20 and input to an encryption process 302 (which is the same encryption process as
21 was used by integrity verification tool 122, or alternatively is a different process
22 that generates the same results given the same inputs) using secret key 310 (the
23 same secret key as was used by integrity verification tool 122). This continues for
24 each block of module 250 (in the same order as was performed by integrity
25 verification tool 122), the output MAC value is obtained from inputting the last

1 block into operator 300. This calculated MAC value is then compared to the
2 original checkpoint value (stored as checkpoint value 270 in module 254). If the
3 two match (are the same), then the integrity of module 250 is verified; otherwise,
4 the integrity of module 250 is not verified.

5 A similar process is used for evaluator 134 to verify the integrity of module
6 254 – a MAC value is generated and compared to the original checkpoint value for
7 module 254 (stored as checkpoint 264 of module 250). If the two match, then the
8 integrity of module 254 is verified; otherwise the integrity is not verified.

9 Although the above process is described with reference to adding additional
10 blocks to each module that is checking the verification of another module, the
11 process may alternatively add an additional block to fewer modules. For example,
12 assume a cyclic verification loop is created where module 250 verifies the
13 integrity of module 254, module 254 verifies the integrity of module 252, and
14 module 252 verifies the integrity of module 250. In this situation, the MAC value
15 for module 250 can be generated and stored in module 254, and the MAC value
16 for module 254 can be generated and stored in module 252. The MAC value for
17 module 252 can then be generated and stored in module 250, and then an
18 additional block added to module 250 to offset the added MAC value (in the
19 manner described above). Thus, with the additional block in module 250, the
20 MAC value that is subsequently generated by evaluator 134 based on module 250
21 will match the MAC value stored in module 254, the MAC value that is
22 subsequently generated by evaluator 134 based on module 254 will match the
23 MAC value stored in module 252, and the MAC value that is subsequently
24 generated by evaluator 134 based on module 252 will match the MAC value
25

1 stored in module 250, even though an additional block has been added to only one
2 of the three modules.

3 In the discussions above, the code integrity verification is described as
4 primarily evaluating the shape of the code (that is, checking the code itself and
5 verifying that it has not been altered). Alternatively, the behavior of the code may
6 be evaluated rather than the shape. Evaluating the behavior of the code refers to
7 evaluating values that change during execution of the code (e.g., dynamic values
8 in certain registers). The generation of the MAC value may be altered to
9 incorporate such values (either in addition to or in place of the shape of the code)
10 to evaluate the behavior of the code during execution to verify that the code is
11 executing properly. For example, instructions in the code may alter a particular
12 register during execution in some known manner and the value(s) in this register
13 incorporated into the MAC value generation process in any of a variety of
14 manners (e.g., by adding the value in the register to the input or output of
15 encryption process 302).

16 Fig. 7 is a flowchart illustrating an exemplary process for generating cyclic
17 checkpoint values in accordance with certain embodiments of the invention. In
18 the illustrated example, the process of Fig. 7 is carried out by checkpoint
19 calculator/insertter 212 (Fig. 3) of integrity verification tool 122 (Fig. 1), and may
20 be performed in software.

21 Initially, the object that is to be protected is separated into multiple modules
22 which are to have their integrity checked (act 350). Multiple blocks are then
23 identified in each module (act 352) and an original MAC value is generated for
24 each module based on the blocks of that module (act 354). The original MAC
25 values are then incorporated into checker modules (act 356).

1 A new MAC value is then generated for each checker module based on the
2 blocks of the checker module (act 358). This new MAC value is based on the
3 blocks of the module after the original MAC value(s) has been added to the
4 module. A new block is then created for each checker module (act 360). For each
5 checker module, the content for the new block in that checker module is
6 determined so as to re-store the MAC value of the checker module to the original
7 MAC value (act 362).

8 9 **Conclusion**

10 Although the description above uses language that is specific to structural
11 features and/or methodological acts, it is to be understood that the invention
12 defined in the appended claims is not limited to the specific features or acts
13 described. Rather, the specific features and acts are disclosed as exemplary forms
14 of implementing the invention.
15
16
17
18
19
20
21
22
23
24
25